Specifying and Proving Properties of Timed I/O Automata in the TIOA Toolkit *

Myla Archer

Center for High Assurance Computer Systems, Code 5546
Naval Research Laboratory, Washington, DC 20375 USA
archer@itd.nrl.navy.mil

HongPing Lim   Nancy Lynch   Sayan Mitra   Shinya Umeno
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139 USA hongping,lynch,mitras,umeno@csail.mit.edu

Abstract

Timed I/O Automata (TIOA) *is a mathematical framework for modeling and verification of distributed systems that involve discrete and continuous dynamics. TIOA can be used for example, to model a real-time software component controlling a physical process. The TIOA model is sufficiently general to subsume other models in use for timed systems. The* TIOA toolkit, *currently under development, is aimed at supporting system development based on TIOA specifications. The TIOA toolkit is an extension of the IOA toolkit, which provides a specification simulator, a code generator, and both model checking and theorem proving support for analyzing specifications. This paper focuses on modeling of timed systems with TIOA and the TAME-based theorem proving support provided in the toolkit for proving system properties, including timing properties. Several examples are provided by way of illustration.*

1 Introduction

To achieve high assurance in the development of complex systems, an appropriate development framework supporting system specification, implementation, and analysis is essential. The support provided by the framework should apply not only to those systems that can be modeled as finite state machines but to those that cannot, such as many real-time embedded or hybrid systems systems involving software and/or continuous behavior. Thus an ideal general development framework should provide:

1. A mathematical model capable of capturing the range of discrete and continuous phenomena that arise in typical systems,

2. A well defined notion in the model of external (visible) behavior, and a definition of implementation of one component by another, or equivalence of two components, in terms of their visible behavior,

3. Compositionality—i.e, the ability to build larger systems by composing smaller components in a manner that respects the notion of implementation,

4. User-friendly tool support for proving the commonly encountered types of properties for the models, such as invariant properties, implementation relations, and stability, and

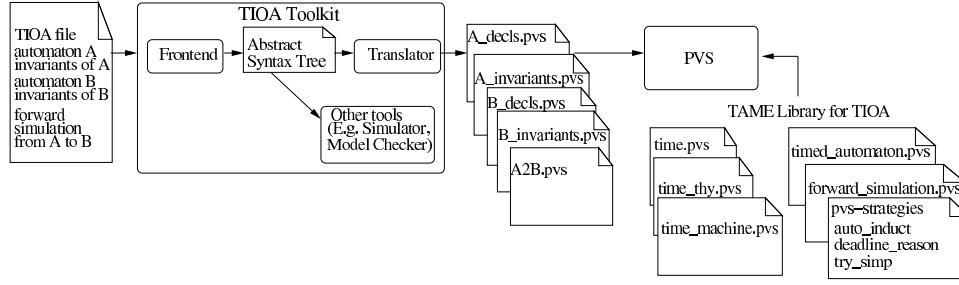5. A basis supporting the use of automatic analysis and other software tools to the extent possible.

The Timed Input/Output Automaton (TIOA) toolkit [15, 9], currently under development, provides just such a framework. The TIOA toolkit, based on the TIOA model [16], is especially suited to the specification and analysis of real-time, embedded systems.

The focus of this paper is on the theorem proving support provided in the TIOA toolkit for the analysis of TIOA specifications. With a set of small examples, we illustrate how one can use the toolkit to model timed systems and specify their properties in the TIOA language, and then verify the specified properties using the theorem prover PVS [28] through the interface TAME [3].

The paper is organized as follows. Section 2 gives an overview of the Timed I/O Automaton (TIOA) model and the TIOA toolkit that supports its use. Section 3 describes how one can specify and prove properties of TIOA models and how the TIOA toolkit supports verifying (or proof checking) the properties mechanically in PVS. Section 4 presents our example TIOA specifications of automata and their properties, and shows how the properties

Report Documentation Page		Form Approved OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **2006**	2. REPORT TYPE	3. DATES COVERED **00-00-2006 to 00-00-2006**
4. TITLE AND SUBTITLE **Specifying and Proving Properties of Timed I/O Automata in the TIOA Toolkit**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Research Laboratory,Center for High Assurance Computer Systems,4555 Overlook Avenue, SW,Washington,DC,20375**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES **10**	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**			

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

TIOA file
automaton A
invariants of A
automaton B
invariants of B
forward
simulation
from A to B

TIOA Toolkit

Frontend

Abstract
Syntax Tree

Translator

Other tools
(E.g. Simulator,
Model Checker)

A_decls.pvs

A_invariants.pvs

B_decls.pvs

B_invariants.pvs

A2B.pvs

PVS

TAME Library for TIOA

time.pvs

time_thy.pvs

time_machine.pvs

timed_automaton.pvs

forward_simulation.pvs

pvs−strategies
auto_induct
deadline_reason
try_simp

_traj_evolve and deadline_reason can be seen in Section 4.1, Figure 6 and Section 4.2, and Figure 13 respectively.

TAME proof step	effect
(apply_traj_evolve t)	Compute state time t from now
(apply_traj_stop t)	Deduce that the stopping condition cannot hold after time t in a trajectory T unless T ends at t
(apply_traj_invariant t)	Deduce trajectory invariant holds time t from now
(deadline_reason t)	Deduce trajectory cannot evolve more than time t if a deadline is reached time t from now

Table 1. New TAME strategies for trajectories.

3 Overview of the TIOA proof methodology

The TIOA mathematical model is useful for specifying timed distributed systems and analyzing properties of the systems as invariants and simulation relations. The model also provides a means of organizing proofs of such properties by induction over the length of the execution of an automaton into a systematic case analysis with respect to the actions and trajectories. It is therefore possible to develop PVS strategies to partially automate such proofs.

The TIOA methodology for theorem proving involves (1) writing the specification of a system and its properties in the TIOA language, (2) using the translator tool to generate the PVS equivalent of the system, and then (3) proving the properties in PVS using TAME strategies (see Figure 1). The user describes the system in the TIOA language using the state-transition structure. The user writes simple program statements to describe transitions, and specifies trajectories using differential equations. Once the TIOA description is type checked by the front end of the toolkit, the translator generates a set of PVS files. Together with the TAME library containing PVS definitions for timed I/O automata and any additional data type theories, these generated files specify the automaton and its properties. The user then uses TAME strategies developed for TIOA to prove the properties of the system in PVS.

By using this approach, the user avoids having to write the automaton description directly in PVS. Moreover, the translator also performs the task of translating program statements in TIOA into functional relations in PVS, and trajectories with differential equations into time-passage actions. An additional benefit gained from using the approach is that the user can also use other tools in the toolkit including the simulator, code generator and model checker.

4 Examples

This section provides three simple examples that together illustrate how TIOA is used to represent systems and properties, how trajectories can be used to capture desired timing behavior, and how system properties can be mechanically verified using PVS. The first example, `fischer`, is a timed version of Fischer's mutual exclusion algorithm. We use this example to illustrate in some detail how various features of a TIOA specification, in particular, its trajectories, are represented in PVS. We also illustrate how its main correctness property, an invariant, can be proved using TAME. The second example, `TwoTaskRace` (representing, as its name suggests, a two task race), is used as an example in which the main correctness property is an abstraction property (forward simulation). The last example, `timeout`, representing a simple timeout system, is used to illustrate the support provided for expressing and reasoning about complex data types in the TIOA toolkit.

4.1 Fischer's mutual exclusion algorithm

Fischer's mutual exclusion algorithm solves the mutual exclusion problem in which multiple processes compete for a shared resource. Figure 2 shows the TIOA specification of a timed version of the Fischer algorithm.

In the Fischer algorithm, each process proceeds through different phases in order to get to the `critical` phase where it gains access to the shared resource. In the automaton used to model the algorithm, each phase has a corresponding action; timing is modeled in the algorithm by time bounds on the actions. The interesting action cases are `test`, `set`, and `check`. The action `set` has an upper time bound, `u_set`, while the action `check` has a lower time bound `l_check`, and `u_set < l_check`. When a process enters the `test` phase, it tests whether the value of a shared variable `x` has been set by any process; if not, the process can proceed to the next phase, `set`, within the upper time bound, `u_set`. In the `set` phase, the process sets a shared variable `x` to its index. Thereafter, the process can proceed to the next phase `check` only after `l_set` amount of time has elapsed. In the `check` phase, the process checks to see if `x` contains the index of the process. If so, it proceeds to the `critical` phase.

The safety property we want to prove is that no two processes are simultaneously in the `critical` phase. We also prove simpler invariants to help us prove this main invariant. Figure 3 shows all the invariants that we have proved, the last invariant being the safety property.

To illustrate how the various elements of an automaton specification in TIOA translate into TAME, Figure 4 shows the TAME specification output by the TIOA-to-TAME translator applied to the TIOA specification in Figure 2. The TAME specification has been edited slightly to save space. In the TAME specification, automaton parameters are translated as constants, and the `where` clause constraining the parameters is expressed as an axiom named `const_facts`. The state variables are represented as a record type named `states`. A `start` predicate is defined to be true for states with the specified initial values. The actions of the automaton are declared as a subset of the `actions` data type in the TAME specification. A predicate `enabled` captures the precondition for each action, while a transition function `trans` captures the post-state obtained by applying the transition of an action on a given pre-state. In translating the effect of an action into the transition function, the translator performs explicit substitutions in accordance with the program statements in the specification of the effect of the action in TIOA, in order to express each state variable in the post-state explicitly in terms of the variables in the pre-state.

The trajectory definition `traj` in the TIOA specification is translated as a time passage action `nu_traj` in the TAME specification which has two parameters:

```
vocabulary fischer_types
  types process,
  PcValue enumeration [ pc_rem, pc_test, pc_set, pc_check,
    pc_leavetry, pc_crit, pc_leaveexit, pc_reset]

automaton fischer(l_check, u_set: Real) where
  u_set < l_check ∧ u_set ≥ 0 ∧ l_check ≥ 0
  imports fischer_types
  signature
    output try(i: process)      internal test(i: process)
    output crit(i: process)     internal set(i: process)
    output exit(i: process)     internal check(i: process)
    output rem(i: process)      internal reset(i: process)
  states
    turn: Null[process] := nil,
    now: Real := 0,
    pc: Array[process, PcValue] := constant(pc_rem),
    last_set: Array[process, AugmentedReal] := constant(u_set),
    first_check: Array[process, Real] := constant(0)
  transitions
    internal test(i)                    internal reset(i)
      pre pc[i] = pc_test                 pre pc[i] = pc_reset
      eff if turn = nil then              eff pc[i] := pc_leaveexit;
            pc[i] := pc_set;                  turn := nil;
            last_set[i] :=
              now + u_set                 output try(i)
          fi                                pre pc[i] = pc_rem
                                            eff pc[i] := pc_test
    internal set(i)
      pre pc[i] = pc_set                  output crit(i)
      eff turn := embed(i);                 pre pc[i] = pc_leavetry
          pc[i] := pc_check;                eff pc[i] := pc_crit
          last_set[i] := \infty;
          first_check[i] :=              output exit(i)
            now + l_check;                  pre pc[i] = pc_crit
                                            eff pc[i] := pc_reset
    internal check(i)
      pre pc[i] = pc_check ∧            output rem(i)
          first_check[i] ≤ now           pre pc[i] = pc_leaveexit
      eff if turn = embed(i) then          eff pc[i] := pc_rem;
            pc[i] := pc_leavetry
          else
            pc[i] := pc_test
          fi;
          first_check[i] := 0;

  trajectories
    trajdef traj
      stop when
        ∃ i: process (now = last_set[i])
      evolve
        d(now) = 1
```

Figure 2. TIOA specification for `fischer`**.**

```
  invariant of fischer:
    ∀ k: process (pc[k] = pc_set ⇒
                    (last_set[k] ≤ (now + u_set)))

  invariant of fischer:
    ∀ k: process (now ≤ last_set[k])

  invariant of fischer:
    ∀ k: process
      (pc[k] = pc_set ⇒ last_set[k] ≠ \infty)

  invariant of fischer:
    ∀ i: process ∀ j: process
      (pc[i] = pc_check
        ∧ turn = embed(i)
        ∧ pc[j] = pc_set
          ⇒ first_check[i] > last_set[j])

  invariant of fischer:
    ∀ i: process ∀ j: process
      (pc[i] = pc_leavetry ∨ pc[i] = pc_crit
        ∨ pc[i] = pc_reset
          ⇒ turn = embed(i) ∧ pc[j] ≠ pc_set)

  invariant of fischer:
    ∀ i: process ∀ j: process
      (i ≠ j ⇒ pc[i] ≠ pc_crit ∨ pc[j] ≠ pc_crit)
```

Figure 3. TIOA invariants for `fischer`**.**

```
fischer_decls : THEORY BEGIN
. . .
l_check: real;  u_set: real
const_facts: AXIOM U_set < l_check AND u_set >= 0 AND l_check >= 0
states: TYPE = [#
  turn: lift[process],
  now: real,
  pc: array[process -> PcValue],
  last_set: array[process -> time],
  first_check: array[process -> real] #]
start(s: states): bool = s=s WITH [
  turn := bottom,
  now := 0,
  pc := (lambda(i_0: process): pc_rem),
  last_set := (lambda(i_0: process): fintime(u_set)),
  first_check := (lambda(i_0: process): 0)]
f_type(i, j: (fintime?)): TYPE = [(interval(i, j))->states]
actions: DATATYPE BEGIN
  nu_traj(delta_t:{t:(fintime?)| dur(t)>=0},
          f:f_type(zero,delta_t)): nu_traj?
  try(i: process): try?
  . . .
  reset(i: process): reset?
END actions
visible?(a:actions): bool =
  try?(a) OR crit?(a) OR exit?(a) OR rem?(a)
timepassageaction?(a:actions): bool = nu_traj?(a)
traj_invariant(a:(timepassageaction?))(s:states):bool =
  CASES a OF nu_traj(delta_t, F): TRUE ENDCASES
traj_stop(a:(timepassageaction?))(s:states):bool = CASES a OF
  nu_traj(delta_t, F):
    EXISTS(i:process): fintime(now(s))=last_set(s)(i)
  ENDCASES
traj_evolve(a:(timepassageaction?)) (t:(fintime?),s:states):states =
  CASES a OF
    nu_traj(delta_t, F): s WITH [now := now(s) + 1 * dur(t)]
  ENDCASES
enabled(a:actions, s:states):bool = CASES a OF
  nu_traj(delta_t, F):
    (FORALL(t:(interval(zero,delta_t))): traj_invariant(a)(F(t)))
    AND
    (FORALL(t:(interval(zero,delta_t))): traj_stop(a)(F(t)) => t=delta_t)
    AND
    (FORALL(t:(interval(zero,delta_t))): F(t)=traj_evolve(a)(t, s)),
  try(i): pc(s)(i) = pc_rem,
  crit(i): pc(s)(i) = pc_leavetry,
  exit(i): pc(s)(i) = pc_crit,
  rem(i): pc(s)(i) = pc_leaveexit,
  test(i): pc(s)(i) = pc_test,
  set(i): pc(s)(i) = pc_set,
  check(i): pc(s)(i) = pc_check AND first_check(s)(i) <= now(s),
  reset(i): pc(s)(i) = pc_reset
  ENDCASES
trans(a:actions, s:states):states = CASES a OF
  nu_traj(delta_t, F): F(delta_t),
  try(i): s WITH [pc := pc(s) WITH [(i) := pc_test]],
  crit(i): s WITH [pc := pc(s) WITH [(i) := pc_crit]],
  exit(i): s WITH [pc := pc(s) WITH [(i) := pc_reset]],
  rem(i): s WITH [pc := pc(s) WITH [(i) := pc_rem]],
  test(i): s WITH [last_set := IF turn(s) = bottom
                                 THEN last_set(s) WITH
                                      [(i) := fintime(now(s) + u_set)]
                                 ELSE last_set(s) ENDIF,
                   pc := IF turn(s) = bottom
                           THEN pc(s) WITH [(i) := pc_set]
                           ELSE pc(s) ENDIF],
  set(i):
    s WITH [turn := up(i),
            last_set := last_set(s) WITH
                [(i) := infinity],
            first_check := first_check(s) WITH
                [(i) := now(s) + l_check],
            pc := pc(s) WITH [(i) := pc_check]],
  check(i):
    s WITH [first_check := first_check(s) WITH [(i) := 0],
            pc := IF turn(s) = up(i)
                    THEN pc(s) WITH [(i) := pc_leavetry]
                    ELSE pc(s) WITH [(i) := pc_test] ENDIF],
  reset(i):
    s WITH [turn := bottom,
            pc := pc(s) WITH [(i) := pc_leaveexit]]
  ENDCASES
IMPORTING timed_auto_lib@time_machine
  [states,actions,enabled,trans,start,visible?,timepassageaction?,
   lambda(a:(timepassageaction?)): dur(delta_t(a))]
END fischer_decls
```

Figure 4. TAME representation of `fischer`

4

```
Inv_5(s:states):bool =
  FORALL (i: process, j: process):
    i /= j => pc(s)(i) /= pc_crit OR pc(s)(j) /= pc_crit

lemma_5: LEMMA FORALL (s:states): reachable(s)=> Inv_5(s);
```

Figure 5. TAME `lemma_5` **for** `fischer`

```
;;; Proof lemma_5-1 for formula fischer_invariants.lemma_5
;;; developed with shostak decision procedures
(""
 (auto_induct)
 (("1" ;; Case nu_traj(delta_t_action, F_action)
   (apply_specific_precond)
   ;; Applying the precondition
   ;; (FORALL (t: (interval(zero, delta_t_action)))):
   ;;    traj_invariant(nu_traj(delta_t_action, F_action))
   ;;                  (F_action(t)))
   ;;   AND
   ;; (FORALL (t: (interval(zero, delta_t_action)))):
   ;;    traj_stop(nu_traj(delta_t_action, F_action))
   ;;             (F_action(t))
   ;;      => t = delta_t_action)
   ;;    AND
   ;; (FORALL (t: (interval(zero, delta_t_action)))):
   ;;     F_action(t) =
   ;;      traj_evolve(nu_traj(delta_t_action, F_action))
   ;;                 (t, prestate))
   (apply_traj_evolve "delta_t_action")
   ;; Using the fact that
   ;; F_action(delta_t_action) =
   ;; prestate WITH
   ;;    [now := 1 * dur(delta_t_action) + now(prestate)]
   (try_simp))
  ("2" ;; Case crit(i_action)
   (apply_specific_precond)
   ;; Applying the precondition
   ;; pc(prestate)(i_action) = pc_leavetry
   (apply_inv_lemma "4" "i_theorem" "j_theorem")
   ;; Applying the lemma
   ;; FORALL (i: process, j: process):
   ;;  pc(prestate)(i) = pc_leavetry OR
   ;;  pc(prestate)(i) = pc_crit OR pc(prestate)(i) = pc_reset
   ;;  => turn(prestate) = up(i) AND pc(prestate)(j) /= pc_set
   (apply_inv_lemma "4" "j_theorem" "i_theorem")
   ;; Applying the lemma
   ;; FORALL (i: process, j: process):
   ;;  pc(prestate)(i) = pc_leavetry OR
   ;;  pc(prestate)(i) = pc_crit OR pc(prestate)(i) = pc_reset
   ;;  => turn(prestate) = up(i) AND pc(prestate)(j) /= pc_set
   (try_simp)))))
```

Figure 6. TAME proof of `lemma_5` **in** `fischer`

`delta_t`, the duration of the trajectory, and `F`, a function representing the trajectory, which maps time values to states. The definitions `traj_invariant`, `traj_stop`, and `traj_evolve` capture the invariant, stopping condition and evolve clause of the trajectory definition respectively. The effect of the "trajectory action" `nu_traj` is constrained—and thus, effectively, captured—by the precondition of `nu_traj`, which asserts that (1) the invariant holds throughout the duration of the trajectory, (2) the stopping condition holds only in the last state of the trajectory, and (3) the evolution of the state variables satisfies the evolve clause. The transition function for `nu_traj` simply returns the post-state obtained by applying the trajectory function `F` after an elapsed time of `delta_t`. This method of representation, adapted from a technique of Luchangco [21], allows `trans` to be represented as a func-

tion from states and actions to states while allowing the result of a `nu_traj` "action" to be nondeterministic.

The new TAME strategies in Table 1, combined with the existing TAME strategies, provide a set of proof steps that allow the `fischer` invariants shown in Figure 3 to be proved interactively in PVS in a clear, high-level fashion. The TIOA-to-TAME translator transforms the six invariants in Figure 3 into TAME invariants and lemmas numbered starting from 0. Thus, the goal safety property, the last invariant in Figure 3, becomes the TAME invariant/lemma pair shown in Figure 5.

Figure 6 shows a verbose TAME proof of `lemma_5` in Figure 5. To create this proof, which can be rerun in PVS, the user simply types in the eight TAME proof steps in the proof script—`(auto_induct)`, `(apply_specific_precond)`, and so on. The comments in this proof (which appear as text after semicolons) are generated by the TAME strategies, and serve to label the proof branches and document the facts introduced by the proof steps in these branches. Because TAME automatically handles "trivial" cases, only the proof steps requiring human guidance need to be recorded. This proof can be understood as follows: The proof step `auto_induct` automates as far as possible the standard initial steps of a proof by induction on the reachable states, including skolemization. The values with names ending in "_theorem" or "_action" are skolem constants standing for variables in the lemma and parameters in the current action, respectively. The name `prestate` refers to the prestate of the current action, and the values of state variables in any state `s` are represented as functions of `s`. The base case and all the action cases except `nu_traj(delta_t_action, F_action)` and `crit(i_action)` are trivial. The `nu_traj(delta_t_action, F_action)` case is proved by recalling the full precondition with `apply_specific_precond`, and then using the new TAME step `traj_evolve` in Table 1 to compute what the current state will be after time `delta_t_action`. Once this is done, only "obvious" reasoning is needed, which is performed by `try_simp`. The proof in the `crit(i_action)` case first recalls the precondition and then uses `apply_inv_lemma` to apply two earlier invariant lemmas to appropriate instances of their quantified variables. Then, only "obvious" reasoning with `try_simp` is needed to complete the proof.

4.2 A two task race

The two-task race system (see Figure 7 for its TIOA description) increments a variable `count` repeatedly, within `a1` and `a2` time, `a1 < a2`, until it is interrupted by a `set` action. This `set` action can occur between `b1` and `b2` time from the start, where `b1 ≤ b2`. After `set`, the value of `count` is decremented (every [a1, a2] time) and a `report` action is triggered when `count` reaches 0.

```
  automaton TwoTaskRace(a1, a2, b1, b2: Real) where
2    a1 > 0 ∧ a2 > 0 ∧ b1 ≥ 0 ∧ b2 > 0 ∧ a2 ≥ a1 ∧ b2 ≥ b1

4    signature
      internal increment
6      internal decrement
      internal set
8      output report
     states
10     count: Int := 0,
       flag: Bool := false,
12     reported: Bool := false,
       now: Real := 0,
14     first_main: Real := a1,
       last_main: AugmentedReal := a2,
16     first_set: Real := b1,
       last_set: AugmentedReal := b2
18   transitions
     internal increment
20     pre ¬flag ∧ now ≥ first_main
       eff count := count + 1;
22         first_main := now + a1;
           last_main := now + a2
24     internal set
       pre ¬flag ∧ now ≥ first_set
26     eff flag := true;
           first_set := 0;
28         last_set := \infty
     internal decrement
30     pre flag ∧ count > 0 ∧ now ≥ first_main
       eff count := count - 1;
32         first_main := now + a1;
           last_main := now + a2
34   output report
       pre flag ∧ count = 0 ∧ ¬reported ∧ now ≥ first_main
36     eff reported := true;
           first_main := 0;
38         last_main := \infty
     trajectories
40     trajdef traj
       stop when now = last_main ∨ now = last_set
42     evolve
         d(now) = 1
```

Figure 7. `TwoTaskRace` **in TIOA**

```
  automaton TwoTaskRaceSpec(a1, a2, b1, b2: Real) where
2    a1 > 0 ∧ a2 > 0 ∧ b1 ≥ 0 ∧ b2 > 0 ∧ a2 ≥ a1 ∧ b2 ≥ b1
     signature
4      output report
     states
6      reported: Bool := false,
       now: Real := 0,
8      first_report: Real :=
         if a2 < b1 then min(b1, a1) + (((b1 - a2) * a1) / a2) else a1,
10     last_report: AugmentedReal :=
         b2 + a2 + ((b2 * a2) / a1)
12   transitions
     output report
14     pre ¬reported ∧ now ≥ first_report
       eff reported := true;
16         first_report := 0;
           last_report := \infty
18   trajectories
     trajdef pre_report
20     invariant ¬reported
       stop when now = last_report
22     evolve
         d(now) = 1
24   trajdef post_report
       invariant reported
26     evolve
         d(now) = 1
```

Figure 8. `TwoTaskRaceSpec` **in TIOA**

We want to show that the time bounds on the occurrence of the `report` action are: lower bound: `if a2 < b1 then min(b1,a1) + ` $\frac{\text{(b1-a2)*a1}}{\text{a2}}$ ` else a1`, and upper bound: `b2 + a2 + ` $\frac{\text{b2*a2}}{\text{a1}}$. This property is proved by specifying an abstract automaton `TwoTaskRaceSpec` which performs a `report` action within these bounds (see Figure 8) and defining a forward simulation relation from `TwoTaskRace` to `TwoTaskRaceSpec` (see Figure 10).

```
TwoTaskRaceSpec_decls : THEORY BEGIN
. . .
% Trajectory invariants
traj_invariant(a:(timepassageaction?))
            (s:states):bool =
  CASES a OF
    nu_pre_report(delta_t,F): NOT reported(s),
    nu_post_report(delta_t,F): reported(s)
  ENDCASES
% Trajectory stopping conditions
traj_stop(a:(timepassageaction?))
        (s:states):bool =
  CASES a OF
    nu_pre_report(delta_t,F):
      fintime(now(s))=last_report(s),
    nu_post_report(delta_t,F):
      true
  ENDCASES
% Trajectory evolve clauses
traj_evolve(a:(timepassageaction?))
          (t:(fintime?),s:states):states =
  CASES a OF
    nu_pre_report(delta_t,F):
      s WITH [now := now(s) + 1 * dur(t)],
    nu_post_report(delta_t,F):
      s WITH [now := now(s) + 1 * dur(t)]
  ENDCASES
% Enabled
enabled(a:actions, s:states):bool = CASES a OF
    nu_pre_report(delta_t,F):
        (FORALL (t:(interval(zero,delta_t))):
          traj_invariant(a)(F(t)))
     AND (FORALL (t:(interval(zero,delta_t))):
          traj_stop(a)(F(t)) => t = delta_t)
     AND (FORALL (t:(interval(zero,delta_t))):
          F(t) = traj_evolve(a)(t, s)),
    nu_post_report(delta_t,F):
        (FORALL (t:(interval(zero,delta_t))):
          traj_invariant(a)(F(t)))
     AND (FORALL (t:(interval(zero,delta_t))):
          traj_stop(a)(F(t)) => t = delta_t)
     AND (FORALL (t:(interval(zero,delta_t))):
          F(t) = traj_evolve(a)(t, s)),
    report: NOT reported(s)
            AND now(s) >= first_report(s)
  ENDCASES
% Transition function
trans(a:actions, s:states):states = CASES a OF
    nu_pre_report(delta_t,F): F(delta_t),
    nu_post_report(delta_t,F): F(delta_t),
    report: s WITH [last_report := infinity,
                    reported := true,
                    first_report := 0]
  ENDCASES
. . .
END TwoTaskRaceSpec_decls
```

Figure 9. `TwoTaskRaceSpec` **trajectories in TAME.**

The abstract automaton `TwoTaskRaceSpec` has two trajectories: `pre_report` and `post_report`. The TAME representation of `TwoTaskRaceSpec` (see Figure 9) illustrates how the translator represents multiple trajectories in TAME: the preconditions in `enabled` and postconditions in `trans` are expressed identically, while the details of the trajectories are captured in separate cases in `traj_invariant`, `traj_stop`, and `traj_evolve`.

The TIOA-to-TAME translator transforms the TIOA specification in Figure 10 of the forward simulation relation into the PVS theory in Figure 11 that asserts (as a theorem to be proved) the property `forward_simulation`. The theory in Figure 11 follows the TAME template

```
forward simulation from TwoTaskRace to TwoTaskRaceSpec:
 ∀ a1: Real ∀ a2: Real ∀ b1: Real ∀ b2: Real
 ∀ last_set: Real ∀ last_main: Real ∀ last_report: Real
 (a1 > 0 ∧ a2 > 0 ∧ b1 ≥ 0 ∧ b2 > 0 ∧ a2 ≥ a1 ∧ b2 ≥ b1
 ∧ last_set≥0 ∧ last_set=TwoTaskRace.last_set
 ∧ last_main≥0 ∧ last_main=TwoTaskRace.last_main
 ∧ last_report≥0 ∧ last_report=TwoTaskRace.last_report
 ⇒ TwoTaskRace.reported = TwoTaskRaceSpec.reported
   ∧ TwoTaskRace.now = TwoTaskRaceSpec.now
   ∧ (¬TwoTaskRace.flag ∧ last_main < TwoTaskRace.first_set
      ⇒ TwoTaskRaceSpec.first_report ≤
        (min(TwoTaskRace.first_set, TwoTaskRace.first_main)
         + ((TwoTaskRace.count
         + ((TwoTaskRace.first_set - last_main) / a2)) * a1)))
   ∧ (TwoTaskRace.flag ∨ last_main ≥ TwoTaskRace.first_set
      ⇒ TwoTaskRaceSpec.first_report ≤
        (TwoTaskRace.first_main + (TwoTaskRace.count * a1)))
   ∧ (¬TwoTaskRace.flag ∧ TwoTaskRace.first_main ≤ last_set
      ⇒ last_report ≥ (last_set + ((TwoTaskRace.count + 2
         + ((last_set - TwoTaskRace.first_main) / a1)) * a2)))
   ∧ (¬(TwoTaskRace.reported) ∧ (TwoTaskRace.flag ∨
        TwoTaskRace.first_main > last_set) ⇒ last_report
        ≥ (last_main + (TwoTaskRace.count * a2))))
```

Figure 10. Forward simulation from `TwoTaskRace` **to** `TwoTaskRaceSpec`

```
TwoTaskRace2TwoTaskRaceSpec: THEORY BEGIN
IMPORTING TwoTaskRace_invariants
IMPORTING TwoTaskRaceSpec_invariants
timed_auto_lib: LIBRARY = "../timed_auto_lib"
MA: THEORY = timed_auto_lib@timed_automaton
            :-> TwoTaskRace_decls
MB: THEORY = timed_auto_lib@timed_automaton
            :-> TwoTaskRaceSpec_decls
amap(a_A: {a: MA.actions |
  visible?(a) AND NOT timepassageaction?(a)}):MB.actions =
  CASES a_A of report: report ENDCASES
ref(s_A: MA.states, s_B: MB.states): bool =
  FORALL (last_set: real, last_main: real, last_report: real):
   a1>0 AND a2>0 AND b1>=0 AND b2>0 AND a2>=a1 AND b2>=b1
   AND last_set >= 0 AND fintime(last_set) = last_set(s_A)
   AND last_main >= 0 AND fintime(last_main) = last_main(s_A)
   AND last_report >= 0 AND fintime(last_report) = last_report(s_B)
   => reported(s_A) = reported(s_B) AND now(s_A) = now(s_B)
    AND (NOT flag(s_A) AND last_main < first_set(s_A) =>
        first_report(s_B) <= min(first_set(s_A), first_main(s_A))
        + count(s_A) + (first_set(s_A) - last_main)/a2*a1)
    AND (flag(s_A) OR last_main >= first_set(s_A) =>
       first_report(s_B) <= first_main(s_A) + count(s_A)*a1)
    AND (NOT flag(s_A) AND first_main(s_A) <= last_set =>
       last_report >= last_set + count(s_A) + 2
                      + (last_set - first_main(s_A))/a1*a2)
    AND (NOT reported(s_A)
        AND (flag(s_A) OR first_main(s_A) > last_set)
        => last_report >= last_main + count(s_A) * a2)
IMPORTING timed_auto_lib@forward_simulation[MA, MB, ref,
 (LAMBDA(a:MA.actions): timepassageaction?(a)),
 (LAMBDA (a:{a:MA.actions|timepassageaction?(a)}):dur(delta_t(a))),
 amap]
fw_simulation_thm: THEOREM forward_simulation
END TwoTaskRace2TwoTaskRaceSpec
```

Figure 11. Simulation relation in TAME

```
invariant of TwoTaskRace:
  a1 >= 0 /\ a2 > 0 /\ b1 >= 0 /\
  b2 > 0 /\ a2 >= a1 /\ b2 >= b1
invariant of TwoTaskRace: now >= 0
invariant of TwoTaskRace: (now + b2) >= 0
invariant of TwoTaskRace: flag => last_set = \infty
invariant of TwoTaskRace: now >= 0 => last_main >= now
```

Figure 12. `TwoTaskRace` **invariants 0–4.**

for formulating abstraction relations between automata described in [26]. The *theory* forward_simulation imported in Figure 11 just before the statement of the theorem provides the generic definition in PVS of the *property* forward_simulation stating what it means for a relation between two automata to be a forward simulation. The PVS formulation of the forward simulation property is

```
;;; Proof lemma_4-1 for formula
;;; TwoTaskRace_invariants.lemma_4
;;; developed with shostak decision procedures
(""
 (auto_induct)
 (("1" ;; Base case
  (const_facts)
  ;; Applying the facts about the constants:
  ;;     a1 > 0 AND a2 > 0 AND b1 >= 0 AND
  ;;     b2 > 0 AND a2 >= a1 AND b2 >= b1
  (try_simp))
 ("2" ;; Case nu_traj(delta_t_action, F_action)
  (apply_specific_precond)
  ;; Applying the precondition
  ;; (FORALL (t: (interval(zero, delta_t_action))):
  ;;     traj_invariant(F_action(t)))
  ;;  AND
  ;; (FORALL (t: (interval(zero, delta_t_action))):
  ;;     traj_stop(F_action(t)) => t = delta_t_action)
  ;;  AND
  ;; (FORALL (t: (interval(zero, delta_t_action))):
  ;;     F_action(t) = traj_evolve(t, prestate))
  (apply_traj_evolve "delta_t_action")
  ;; Using the fact that
  ;; F_action(delta_t_action) =
  ;; prestate WITH
  ;;   [now := 1 * dur(delta_t_action) + now(prestate)]
  (apply_inv_lemma "1")
  ;; Applying the lemma
  ;; now(prestate) >= 0
  (deadline_reason "last_main(prestate)")
  ;; Reasoning that time cannot pass beyond
  ;; last_main(prestate)
  (try_simp))
 ("3" ;; Case increment
  (const_facts)
  ;; Applying the facts about the constants:
  ;;     a1 > 0 AND a2 > 0 AND b1 >= 0 AND
  ;;     b2 > 0 AND a2 >= a1 AND b2 >= b1
  (try_simp))
 ("4" ;; Case decrement
  (const_facts)
  ;; Applying the facts about the constants:
  ;;     a1 > 0 AND a2 > 0 AND b1 >= 0 AND
  ;;     b2 > 0 AND a2 >= a1 AND b2 >= b1
  (try_simp))
 ("5" ;; Case report
  (try_simp))))
```

Figure 13. Proof of `TwoTaskRace` **invariant 4.**

based on the definition in [24]. The proof of this property for `TwoTaskRace` and `TwoTaskRaceSpec` uses invariants of both automata.

The invariants of `TwoTaskRace` and `TwoTaskRace-Spec` needed for the forward simulation proof have all been proved in TAME. The proofs of these invariants are all quite simple; in fact, all of the invariants needed for `TwoTaskRaceSpec` are proved automatically by the TAME induction strategy auto_induct. The proofs of a few of the invariants for `TwoTaskRace` are interesting because they illustrate the use of the new TAME strategy deadline_reason, which was not used in the invariant proofs for fischer. One such invariant is invariant 4 in Figure 12, whose TAME proof is shown in Figure 13. Invariant 4 essentially says that in the TIOA model of `TwoTaskRace`, the current time now cannot pass beyond the deadline last_main. In this proof, auto_induct has determined that the base case and four of the five possible action cases are nontrivial. The crux of this proof

is the reasoning in the single time passage case, namely, the action case `nu_traj(delta_t_action)`. After using `apply_specific_precond` and `apply_traj_e-volve` to compute the state after time `delta_t_action` and using `apply_inv_lemma` to use invariant 1 to establish that `now >= 0` at the beginning of the trajectory, the new TAME step `deadline_reason` argues that `now <= last_main` at the end of the trajectory. The step `try_simp` then completes the proof with "obvious reasoning". The remaining cases are easily proved using "obvious reasoning" following, in some cases, the use of `const_facts` to introduce facts about the constants in the specification.

TAME also provides strategies for establishing abstraction relations between automata, including forward simulation. Forward simulation proofs have a high-level structure similar to the structure of induction proofs of invariants; however, rather than beginning with the proof step `auto_induct`, they begin with the proof step `prove_fwd_sim`. For more details, see [26].

4.3 A simple timeout system

A simple timeout system consists of a sender, a delay prone channel, and a receiver (see Figure 14 for its TIOA description). The sender sends messages to the receiver, within `u1` time after the previous message has been sent. A `timed_message_Queue` delays the delivery of each message by at most `b` time. A failure can occur at any time,

after which the sender stops sending. The receiver times out after not receiving a message for at least `u2` time.

We are interested in proving the two following properties for this system: (1) Safety: A timeout occurs only after a failure has occurred; (2) Timeliness: A timeout occurs within `u2 + b` time after a failure. The safety property can be captured by an invariant of the system. As in the two-task race example, to show the timeliness, we first create an abstract automaton that times out within `u2 + b` time of occurrence of a failure, and then we prove a forward simulation from the system to its abstraction. Both the safety and timeliness properties have been proved using the TAME strategies in a manner analogous to the invariant and for-

```
vocabulary timed_queue
 types M, timed_message_Queue, timed_message
 operators
   mtQ: -> timed_message_Queue
   enQ_qn: timed_message_Queue -> Bool
   deQ: timed_message_Queue -> timed_message_Queue
   enQ: timed_message, timed_message_Queue
            -> timed_message_Queue
   MKtimed_message: M, Real -> timed_message
   earliest_msg: timed_message_Queue -> M
   earliest_deadline: timed_message_Queue
                       -> AugmentedReal
   latest_deadline: timed_message_Queue -> Real
   time_ordered: timed_message_Queue -> Bool
   nthQ: timed_message_Queue, Nat -> M
   lengthQ: timed_message_Queue -> Nat
   deadline: M -> Real
```

Figure 15. TIOA declaration of custom data types and operators used in `timeout`.

```
Queue[T:TYPE]: DATATYPE
 BEGIN
  mtQ: mtQ?
  enQ(last:T, before_last:Queue): enQ?
 END Queue
Queue_thy[T:type]: THEORY
 BEGIN
  IMPORTING Queue[T]
  lengthQ(q:Queue): RECURSIVE nat =
   IF mtQ?(q) THEN 0
   ELSE lengthQ(before_last(q)) + 1 ENDIF
  MEASURE reduce_nat(0, (LAMBDA (x:T), (n:nat): n+1));
  deQ(q:(enQ?)): RECURSIVE Queue =
   IF mtQ?(before_last(q)) THEN mtQ
   ELSE enQ(last(q),deQ(before_last(q))) ENDIF
  MEASURE lengthQ(q);
  nthQ(q:(enQ?),
       n:{i:nat| 0<=i & i<=lengthQ(q)-1}): RECURSIVE T =
   IF n=0 THEN last(q) ELSE nthQ(before_last(q),n-1) ENDIF
  MEASURE n;
  . . .
 END Queue_thy
timed_message_Queue_thy[M:TYPE] : THEORY
 BEGIN
  . . .
  IMPORTING timed_message_thy[M]
  IMPORTING Queue_thy[timed_message]
  timed_message_Queue: TYPE = Queue[timed_message];
  time_ordered(q:timed_message_Queue): bool =
   FORALL (i: [upto(lengthQ(q) - 1)],
       j: {n:nat | n >= i & n <= lengthQ(q)-1}):
   deadline(nthQ(q,i)) >= deadline(nthQ(q,j));
 END timed_message_Queue_thy
```

Figure 16. Sample PVS definitions of custom data types and operators used in `timeout`.

```
   automaton timeout(u1, u2, b: Real)
 2   where u1 ≥ 0 ∧ u2 ≥ 0 ∧ b ≥ 0 ∧ u2 > (u1 + b)
     imports timed_queue
 4   signature
       internal send(m: M)
 6     internal receive(m: M)
       output fail
 8     output timeout
     states
10     p_clock: AugmentedReal := 0,
       t_clock: AugmentedReal := u2,
12     suspected: Bool := false,
       failed: Bool := false,
14     now: Real := 0,
       queue: timed_message_Queue := mtQ
16   transitions
     internal send(m)
18     pre now ≥ 0 ∧ ¬failed ∧ p_clock = now
       eff if (now + u1) ≥ 0 then p_clock := now + u1 fi;
20         if (now + b) ≥ latest_deadline(queue) then
               queue := enQ(MKtimed_message(m, now + b), queue)
22         fi;
     internal receive(m)
24     pre now ≥ 0 ∧ enQ_qn(queue) ∧ m = earliest_msg(queue)
       eff if (now + u2) ≥ 0 then t_clock := now + u2 fi;
26         if enQ_qn(queue) then queue := deQ(queue) fi
     output fail
28     pre ¬ failed
       eff failed := true;
30         p_clock := \infty
     output timeout
32     pre now ≥ 0 ∧ ¬suspected ∧ t_clock = now
       eff suspected := true;
34         t_clock := \infty
   trajectories
36     trajdef traj
       stop when now ≥ 0 ∧ (now = p_clock ∨ now = t_clock
38             ∨ now = earliest_deadline(queue))
       evolve d(now) = 1
```

Figure 14. TIOA description of `timeout`

ward simulation proofs in the previous examples, with one extra complication: the need to introduce knowledge about special data types referred to in the TIOA specifications.

The timeout system makes use of a custom data type `timed_message_queue`. TIOA provides a `vocabulary` syntax to allow the user to declare custom data types and operators. Figure 15 shows how the data type for `timed_message_queue` and the associated operators are declared in TIOA. The actual PVS definitions of these types and operators are provided as part of a TIOA library of data type theories; Figure 16 shows a sample of these definitions. Aside from the PVS operator `enQ?` (which implements the TIOA operator `enQ_qn` for querying whether a `timed_message_queue` is a nonempty queue), the PVS vocabulary is identical to the TIOA vocabulary. Properties of these data types have been proved in PVS, and have been used in proofs of the specification properties.

5 Discussion

Developing theorem proving support. Our approach to developing appropriate theorem proving support for TIOA is to study many examples of TIOA specifications and their properties and identify what is needed for implementing a standard, straightforward set of proof steps sufficient to mechanize proofs of the properties. One lesson we have learned is that the details of the specification template that a translator to PVS targets, if chosen carefully, can greatly facilitate the implementation of PVS strategies. Details of the TAME template for TIOA that have proved helpful for strategy development include the overall scheme for representing trajectories illustrated in Figure 9 and the scheme for representing the start state predicate `start(s)` as an equality of the form `s = ...`, possibly in conjunction with additional restrictions (see, for example, Figure 4). Another detail of our translation scheme is the use of symbolic computation, if necessary, to permit the effects of transitions, which are defined in TIOA as the effect of a sequence of computations, to be represented in `trans` by explicit updates to state variables. This allows the theorem prover to reason directly about new state values of individual variables with less effort.

One goal in developing support for interactive theorem proving is to find a minimal set of proof steps that are natural to use in high level reasoning and that are sufficient (or nearly so) for mechanizing proofs of properties. Studying many examples has helped us in this regard. For example, we observed that many proofs included the observation that time cannot pass beyond a given deadline unless some discrete action occurs. This observation led us to include `deadline_reason` among our set of proof steps.

Mechanizing proofs. The theorem proving support we are developing for TIOA does not make mechanizing proofs of properties automatic, but it does make it simpler. A user who wishes to prove properties of a TIOA specification using TAME must in general be a domain expert for the system modeled in TIOA. To prove the desired safety or simulation properties, the user often must first find an appropriate set of supporting lemmas. Doing this may require some creativity; some guidance on how to go about it can be found in [24]. The user must also be able to sketch out at a high level why, based on the set of supporting lemmas, a given property is expected to hold. To produce a mechanical proof of the property, the user then can apply TAME reasoning steps that match this high level reasoning. Typically, this can be done using steps such as `const_facts`, `apply_inv_lemma`, `apply_specific_precond`, `deadline_reason`, and so on, to introduce the facts appealed to in each nontrivial case in the proof sketch, and then invoking `try_simp` to do the "obvious" reasoning based on these facts.

While it is good to have a mechanical check of a proof's validity, it is equally important to have some feedback on what went wrong if the mechanical check fails. For failed proofs, TAME provides some useful feedback: the saved TAME proof script can be used to detect the place in the proof where the proof breaks down. The user can then review the high level reasoning to see whether there is an error or if introducing additional facts can complete the proof.

Scalability. We have begun experimentation with using the TAME support for TIOA on larger examples. Our first larger example is the Small Aircraft Traffic System protocol `SATS` developed at NASA Langley. An abstract model of this system has been defined in [8]. An IOA version of this model has been represented and verified in PVS [29]. We have used the TIOA-to-TAME translator to represent the IOA model in TAME, and have begun redoing the proofs using the TAME strategies.

The `SATS` example has raised an issue that is likely to arise in many large examples: the use by specifiers of multi-layered definitions of application-specific functions and predicates. One way to manage the many definition expansions for proof efficiency would be to expand them in layers to allow reasoning to proceed at the highest possible layer. A goal for the translator is to generate "local strategies" for a specific application that group definitions by layer. A scheme of this sort is used in the SCR-to-TAME translator to increase the efficiency of the TAME strategies that support reasoning about SCR automata [3].

6 Related work

Previous work has been performed to develop tools to translate specifications written in the IOA language to the language of various theorem provers, for example, Larch [6, 10], PVS [7], and Isabelle [30, 27]. Our implementation of the TIOA to PVS translator described in [20] builds upon [6]. The target PVS specifications of this trans-

lator strongly resemble TAME specifications. In addition, an early version of TAME's `deadline_reason` strategy was implemented as the PVS strategy `deadline_check` described in [20]. The TIOA-to-TAME translator is essentially a version of the TIOA-to-PVS translator of [20] with modifications that allow the straightforward implementation of new TAME strategies for TIOA and the most effective use of existing TAME strategies. A more complete description of the recent improvements made to the translation scheme and strategies described in [20] can be found in [19]. In [12], a slightly different approach using *urgency predicates* instead of stopping conditions or invariants to limit trajectories is used to describe timed I/O automata. An approach to proving invariant properties of timed I/O automata using urgency predicates is described, but no tool support. A proposed design for supporting urgency predicates in the TIOA toolkit is given in [4].

7 Conclusion

The TIOA framework is ultimately intended to support all phases of system development from specification, through verification and validation, to implementation. In this paper, we have focused on the usability of the TIOA framework for modeling and mechanical verification of properties of timed systems with both discrete and continuous transitions. We have described the theorem proving support provided, and illustrated how it is used in examples where the properties of interest are invariant properties or simulation properties, and where the models involve nontrivial data types.

Our plan for the future is experiment with more complex examples, such as SATS or the Dynamic Host Configuration Protocol DHCP (using models based on the work described in [13]), to explore extensions and improvements to our proof support.

References

[1] R. Alur. Timed automata. In *Proc. 11th Intern. Conf. on Computer Aided Verif. (CAV '99)*, volume 1633 of *Lect. Notes in Comp. Sci.*, pages 8–22. Springer-Verlag, 1999.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] M. Archer. TAME: Using PVS strategies for special-purpose theorem proving. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):139–181, 2000. Published Feb. 2001.

[4] M. Archer. Basing a modeling environment on a general purpose theorem prover. In *Proc. Monterey Wkshp. on Soft. Eng. Tools: Compat. and Integr.*, Baden, Austria, Oct. 2004. To appear.

[5] M. Archer, C. Heitmeyer, and E. Riccobene. Proving invariants of I/O automata with TAME. *Automated Software Engineering*, 9(3):201–232, 2002.

[6] A. Bogdanov, S. Garland, and N. Lynch. Mechanical translation of I/O automaton specifications into first-order logic. In *Form. Tech.s for Networked and Distr. Sys. - FORTE 2002 : 22nd IFIP WG 6.1 Intern. Conf.*, pages 364–368, Texas, Houston, USA, Nov. 2002.

[7] M. Devillers. Translating IOA automata to PVS. Technical Report CSI-R9903, Computing Science Institute, University of Nijmegen, February 1999.

[8] G. Dowek, C. Muñoz, and V. Carreño. Abstract model of the SATS concept of operations: Initial results and recommendations. Technical Report NASA/TM-2004-213006, NASA Langley Res. Ctr., Hampton, VA, 2004.

[9] S. Garland. TIOA User Guide and Reference Manual. Technical report, MIT CSAIL, Cambridge, MA, 2006.

[10] S. Garland and J. Guttag. A guide to LP, the Larch prover. Technical report, DEC Systems Research Center, 1991. URL http://nms.lcs.mit.edu/Larch/LP.

[11] S. Garland, N. Lynch, J. Tauber, and M. Viziri. IOA User Guide and Reference Manual. Technical Report MIT-LCS-TR-961, MIT CSAIL, Cambridge, MA, 2004.

[12] B. Gebremichael and F. W. Vaandrager. Specifying urgency in timed I/O automata. In *Proc. 3rd IEEE Intern. Conf. on Softw. Eng. and Form. Meths. (SEFM 2005)*, pages 64–73, Koblenz, Germany, September 5-9 2005. IEEE Comp. Soc.

[13] N. D. Griffeth and C. Djouvas. Experimental method for testing networks. In *Soft. Eng. Research and Practice*, pages 935–941, 2005.

[14] C. Heitmeyer, M. Archer, R. Bharadwaj, and R. Jeffords. Tools for constructing requirements specifications: The SCR toolset at the age of ten. *Intern. J. on Computer System Science and Engineering*, 20(1):19–35, January 2005.

[15] D. Kaynar, N. A. Lynch, R. Segala, and F. Vaandrager. A mathematical framework for modeling and analyzing real-time systems. In *The 24th IEEE Intern. Real-Time Systems Symposium (RTSS)*, Cancun, Mexico, December 2003.

[16] D. Kaynar, N. A. Lynch, R. Segala, and F. Vaandrager. *The Theory of Timed I/O automata.* Synthesis Lectures on Computer Science. Morgan Claypool Publishers, 2005.

[17] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Intern. J. on Software Tools for Tech. Transfer*, 1(1-2):134–152, 1997.

[18] H. Lim. Translating timed I/O automata specifications for theorem proving in PVS. Master's thesis, MIT, Cambridge, MA, 2006.

[19] H. Lim and M. Archer. Translation templates to support strategy development in PVS. In *STRATEGIES06, 6th International Workshop on Strategies in Automated Deduction*, Seattle, USA, August 2006.

[20] H. Lim, D. Kaynar, N. Lynch, and S. Mitra. Translating timed I/O automata specifications for theorem proving in PVS. In *Formal Modeling and Analysis of Timed Systems (FORMATS)*, pages 17–31, Uppsala, Sweden, Sept. 2005.

[21] V. Luchangco. Personal communication. 1996.

[22] N. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, Sept. 1989. Centrum voor Wiskunde en Informatica, Amsterdam, Netherlands.

[23] N. Lynch and F. Vaandrager. Forward and backward simulations – Part II: Timing-based systems. *Information and Computation*, 128(1):1–25, July 1996.

[24] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.

[25] M. Merritt, F. Modugno, and M. R. Tuttle. Time constrained automata. In J. C. M. Baeten and J. F. Goote, eds., *CONCUR'91: 2nd Intern. Conference on Concurrency Theory*, vol. 527 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1991.

[26] S. Mitra and M. Archer. PVS strategies for proving abstraction properties of automata. *Electronic Notes in Theor. Comp. Sci.*, 125(2):45–65, 2005.

[27] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 1994.

[28] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert. PVS Prover Guide, Version 2.4. Technical report, Comp. Sci. Lab., SRI Intl., Menlo Park, CA, Nov. 2001.

[29] S. Umeno and N. Lynch. Proving safety properties of an aircraft landing protocol using I/O Automata and the PVS theorem prover: A case study. To appear as an MIT Technical Report.

[30] T. N. Win. Theorem-proving distributed algorithms with dynamic analysis. Master's thesis, MIT Dept. of Electr. Eng. and Comp. Sci., May 2003.